PUGACE, A Cellular Evolutionary Algorithm framework on GPUs

Nicolás Soca, José Luis Blengio, Martín Pedemonte and Pablo Ezzatti

***Abstract*— Metaheuristics are used for solving optimization problems since they are able to compute near optimal solutions in reasonable times. However, solving large instances it may pose a challenge even for these techniques. For this reason, metaheuristics parallelization is an interesting alternative in order to decrease the execution time and to provide a different search pattern. In the last years, GPUs have evolved at a breathtaking pace. Originally, they were specific-purpose devices, but in a few years they became general-purpose shared memory multiprocessors. Nowadays, these devices are a powerful low cost platform for implementing parallel algorithms. In this paper, we present a preliminary version of PUGACE, a cellular Evolutionary Algorithm framework implemented on GPU. PUGACE was designed with the goal of providing a tool for easily developing this kind of algorithms. The experimental results when solving the Quadratic Assignment Problem are presented to show the potential of the proposed framework.**

I. Introduction

Exact algorithms can be useless for solving large dimension optimization problems in reasonable times due to their high computational complexity. In this context, when the use of exact algorithms is impractical, metaheuristic techniques have emerged as flexible and robust methods for solving optimization problems. Metaheuristic methods do not guarantee to obtain an optimal solution, but they find near optimal solutions efficiently. Evolutionary Algorithms (EAs) is a metaheuristic technique that has become popular in the last twenty years.

Although metaheuristics are efficient techniques, when applied to solve large problem instances they can lead to a significant increase in the execution time. For this reason, metaheuristics parallelization is an interesting alternative in order to decrease the execution time, using the increasing possibilities offered by modern hardware architectures. Parallel metaheuristics can also benefit from using a different search pattern, even improving over traditional sequential methods. Cellular Evolutionary Algorithms (cEAs) are a class of inherently parallel evolutionary algorithms in which the population is structured in neighborhoods and individuals can only interact with their neighbors.

In recent years, Graphics Processing Units (GPUs) have been progressively and rapidly advancing from specialized fixed-function graphics to highly programmable and parallel computing devices. With the introduction of the Compute Unified Device Architecture (CUDA), GPUs are no longer exclusively programmed using graphical APIs. Now, GPUs are exposed to the programmer as a set of general-purpose shared-memory Single Instruction Multiple Data multi-core processors. The number of threads that can be executed in parallel on such devices is currently in the order of hundreds and it is expected to multiply soon.

Nicolás Soca, José Luis Blengio, Martín Pedemonte and Pablo Ezzatti are with the Centro de Cálculo - Instituto de Computación, Universidad de la República, J. H. Reissig 575, Montevideo, Uruguay (phone: +598-2 711 42 44; email: {nsoca,jblengio,mpedemon,pezzatti}@fing.edu.uy).

This paper presents a preliminary version of PUGACE, a Cellular Evolutionary Algorithm framework implemented on a GPU.

The paper is organized as follow. The next section gives a background on graphics hardware. Section III briefly describes EAs and cEAs, and it reviews the related work about evolutionary techniques on GPU. Then, the proposed framework is commented in section IV. Section V presents the experimental analysis, reporting the quality and the performance results for a test problem. The paper ends with the conclusions of the research and suggestions for future work in Section VI.

II. GPUs

In the last years, there was an uprise of hardware accelerators like graphics processors. This growth has been mainly based on two key issues: the GPUs architecture is intrinsically parallel, in contrast to CPUs serial based architecture; and game industry has forced to increase the graphic processors capabilities to make faster and more realistic games.

Old graphics cards used to have a fixed graphics pipeline, that is to say the operations and the order in which they are applied over data were preconfigured. In the last ten years, GPUs have impressively changed. The major key changes are commented next.

Concerning to the programming capabilities, at first GPU only provided their own transformations and lighting operations (vertex shaders) to be performed on vertices, and their own pixel shaders to determine the final pixels color. Whereas now, GPUs have many multiprocessors that can be programmed for general purpose computing.

With respect to accuracy, GPUs originally worked with 8-bits numbers and then switched to the 16-bits floating point format. Subsequently, they supported the single precision floating point system (32 bits); and currently GPUs fully support the standard IEEE double precision arithmetic (64 bits).

Regarding the GPUs programming tools, the shaders originally had to be written in assembly language. With the constant rising of functionality provided by GPUs, different higher level programming languages were developed, such as High-Level Shading Language (HLSL) and nVidia's Cg [1]. Another alternative was to directly use computer graphics tools, such as OpenGL [2] or DirectX [3]. Later, other high level languages emerged based in turning GPUs into a stream processor, such as Brook [4], Sh [5], PyGPU [6], Accelerator

978-1-4244-8126-2/10/$26.00 ©2010 IEEE

languages [7], CTM and ATI Stream Technology [8]. Finally, CUDA [9] and OpenCL [10] were developed.

Compute Unified Device Architecture (CUDA) is the API from nVidia for exposing the processing features of the G80 GPU. CUDA is a C language API that provides services ranging from common GPU operations in the CUDA library to traditional C memory management semantics in the CUDA runtime and device driver layers. Additionally, nVidia provides a specialized C compiler to build programs developed for the GPU.

As it was mentioned before, current GPUs can be considered as shared memory multi-core processors. The processing is based on hundreds of threads that are grouped into blocks. Memory hierarchy is an important attribute of modern GPUs. Nowadays, GPUs have four levels of memory: registers, shared block memory, local memory and global memory. Registers are the fastest memory on the multiprocessor and are only accessible by each thread. Shared memory is almost as fast as registers and could be accessed by any thread of the block. The local and the global memories are the slowest memories on the multiprocessor (they are more than 100 times slower than the shared memory), and while the local memory is only accessible to each thread, the global memory is accessible to all the threads on the GPU.

Information about general-purpose computation on GPUs can be found in Owens et al. [11]. The continuous development of the area can be followed on the GPGPU organization website [12].

III. Cellular Evolutionary Algorithms

This section presents a description of EAs and cEAs. After that, it summarizes the related previous work on the implementation of evolutionary techniques on GPUs.

A. Evolutionary Algorithms

Evolutionary algorithms are stochastic search methods inspired by the natural process of evolution of species. EAs iteratively evolve a population of individuals representing candidate solutions of the optimization problem. The evolution process is guided by a *survival of the fittest* applied to the candidate solutions and it involves the probabilistic application of operators to find better solutions.

The initial population is randomly generated. Each iteration is divided in four stages. In the first stage, every individual in the population is associated with a fitness value that measures the quality of the candidate solution. Afterward, the solutions are selected from the population based on their fitness value, usually giving higher priority to higher quality solutions. In the third stage, new solutions are constructed applying evolutionary operators to the selected solutions. Typically, the evolutionary operators used are the crossover (recombination of parts of two individuals) and the mutation (random changes in a single individual). Finally, in the fourth stage, the new population is created, by replacing the worse adapted individuals with solutions generated in the iteration.

B. Cellular Evolutionary Algorithms

There are different proposals to structure the EA population. Single population or panmixia works with a single population in which there are no group structures and therefore any individual could be mated for reproduction with anyone of the population. On the other hand, in the island model or distributed EAs, the population is split in several subpopulations called islands. Each subpopulation works independently and some solutions are exchanged among them with a given frequency.

Another alternative is known as cEAs [13]. cEAs work with a single population structured in many small overlapped neighborhoods. Each individual is placed in a *cell* on a toroidal n-dimensional grid. Each individual belongs to several neighborhoods, but it has its own neighborhood for reproduction. A given individual can only be mated for reproduction with individuals of its neighborhood. The effect of finding high-quality solutions gradually spreads to other neighborhoods along the grid using a diffusion model as a consequence of the neighborhoods overlapping. Figure 1 presents an example of a cEA population on a 2D grid.

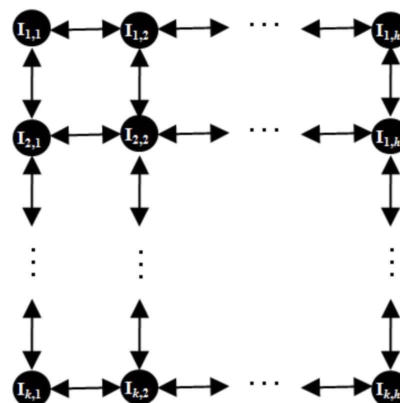

Fig. 1. cEA population on a 2D grid.

C. Related work: evolutionary computation on GPUs

This subsection reviews some of the preceding effort in the design of parallel implementations of EAs on GPUs. This is a relatively new approach and there are few studies regarding the issue. So, the survey covers not only the works that have proposed parallel EA in GPU, but also other evolutionary computation (EC) techniques. In recent times, several authors have presented different interesting contributions about EC on GPUs; most of the work in this area can be found at the GPGPGPU website [14].

One of the pioneering works in this area is the proposal of Wong et al. [15], where the authors studied an Evolutionary Programming (EP) algorithm for solving five simple test functions. In their proposal, denominated Fast Evolutionary Programming (FEP), the selection process takes place in the CPU while the fitness function evaluation and the mutation are performed in the GPU, following a master-slave parallel

model. The authors noticed that the population should be stored in the device memory to obtain high performance values due to the low ratio in the data transfer between CPU and the device memory. For this reason, FEP uses textures to store the population and one additional texture to store the fitness values of all the population individuals. The experimental evaluation was performed on a Pentium IV at 2.4 GHz and a GeForce 6800 Ultra card, comparing the parallel FEP and a sequential implementation. The results showed that speedup values ranged between $0.62\times$ and $5.02\times$ were obtained depending on the population size and the problem complexity. Later [16], the authors proposed a Parallel Hybrid Genetic Algorithm (HGA) on GPU, where the whole evolutionary process is executed on the GPU while the generation of random numbers runs on the CPU. HGA is a Genetic Algorithm that incorporates a Cauchy mutation operator. The authors compared their proposal against a full CPU implementation and the FEP method on an AMD Athlon 64 3000 with 1 GB of memory RAM and a GeForce 6800 Ultra graphics card with 256 MB. The results showed that HGA reaches speedup values between $1.14\times$ and $4.24\times$ when considering population sizes between 400 and 6400 individuals. Finally, in a later work [17] they extended the experimental evaluation considering more difficult instances, reaching speedup values of $5.50\times$.

Harding and Banzhaf [18] implemented a Genetic Programming (GP) algorithm for evolving computer programs represented as tree structures following a master-slave model on GPU. In their proposal, the GPU is only used to evaluate the fitness function. Several problems were considered involving different types of fitness functions such as floating point, boolean expressions and a real world test. The purpose of the conducted tests was to compare the time required for evaluating a tree with a given size for a fixed number of fitness cases in the CPU and in the GPU. The experimental platform was an Intel Centrino T2400 and an nVidia GeForce 7300 GO graphic card. The results showed significant reductions on the time required for evaluating the fitness function on the GPU, with speedup values in the order of tens for real world tests, in the order of hundreds for boolean expressions and in the order of thousands for floating point type when compared against the sequential algorithm implemented in a CPU. Following a similar idea, the authors [19] studied a GP variant to generate shader programs to obtain image filters for noise reduction, but the work focuses on the specific details for solving the problem.

Also following a master-slave model, Maitre et al. [20] extended EASE (a metalanguage to implement EAs) to automatically exploit the GPU capabilities. The proposal was validated by solving two cases: the Weierstrass-Mandelbrot problem and a real world chemistry problem of structure determination. The experimental evaluation was aimed at studying the scalability of the proposal, considering only the execution time corresponding to the fitness function evaluation. The execution platform was a standard 3.6 GHz PC. Two different graphic cards were considered in the experiments, an nVidia 8800 GTX and an nVidia GTX 260. In the Weierstrass-Mandelbrot problem, when using a population of 4096 individuals the speedup of evaluation stage reached values of $33\times$ and $100\times$ on the 8800 GTX and on the GTX 260, respectively. In the real world chemistry problem, when considering a population size of $20,000$ individuals and using an initial codification, the fitness function evaluation stage took 23 s. in the CPU implementation and 80 s. in the GPU implementation. However, the authors were able to reduce the execution time to 7.66 s. for the CPU implementation and 0.33 s. for the GTX 260 implementation by changing the data structures used.

The first work on implementing a cellular EAs using GPUs was made by Yu et al. [21]. The authors studied the Colville minimization problem, placing the individuals in a toroidal 2D grid and using the classical Von Neumann neighborhood structure with five cells. The individuals were represented using a set of texture maps in two dimensions. Each chromosome was divided into several segments, which were assigned to different textures in the same position. Each segment contained four genes packed into each pixel as RGBA color space. An additional texture was used to store the fitness value of each individual. The experiments performed on a computer AMD Athlon2500 and an nVidia GeForce 6800 GT card showed speedup values of up to $20\times$ for populations of 512^2 individuals.

In a similar line of work, Li et al. [22] proposed a cEA completely implemented on GPU for solving the Schwefel, Shaffer and Camel functions. The execution platform was a PC with a Pentium IV 2.66 GHz with 256 MB of RAM memory and an nVidia GeForce 6800 card. This proposal obtained speedup values ranged between $1.4\times$ and $5.4\times$ for a population of sizes between 200 and 800, speedup values of $9.6\times$ to $16.9\times$ for populations with $3,200$ individuals and speedup values ranged between $21\times$ and $73\times$ for a population with $10,000$ individuals.

The proposal by Li et al. [23] follows a different line within the fine-grained parallelism on GPU. The authors studied an immune strategy algorithm applied to GA for solving the Travelling Salesman Problem. The execution platform was an AMD 3000 + with a 9600 GT card, and the speedup values obtained ranged between $2.42\times$ and $11.5\times$.

The island model is the parallelization strategy more lately implemented on GPU. Tsutsui and Fujimoto [24] presented a parallel island model implementation of a GA for solving the Quadratic Assignment Problem. The execution platform was a computer with an Intel i7 965 processor with an nVidia GTX 285 card. The authors obtained speedup values between $3\times$ and $12\times$ when using population of sizes between $3,840$ and $19,200$ for solving instances with up to 40 locations.

Another proposal of a parallel island model was made by Lewis et al. [25], who studied the technique known as genetic cyclic programming. In this work, the authors studied a full GPU implementation, an hybrid GPU-CPU implementation and an hybrid GPU-CPU implementation with two CPUs and two GPUs. The reported results are really

impressive, reaching speedup values in the order of hundreds. The paper also presents an interesting discussion on how to evaluate the performance when comparing GPU and CPU implementations.

Finally, up to our knowledge, Wong [26] presented the only proposal of Multiobjective Evolutionary Algorithms (MOEAs) using GPUs. The author proposed an algorithm inspired on the NSGA-II implemented with CUDA. Two different strategies to implement the non-dominance checking in GPU were discussed. This is an important operation in MOEAs, since it is used to quantify the quality of the solutions. The experimental evaluation was performed in a computer with an Intel Pentium Dual E2220 processor with 2GB RAM and a GeForce 9600 GT graphics card with 512 MB of RAM. The author reported values of speedup between $5.65\times$ and $10.75\times$ when using population sizes between $4,096$ and $16,384$ for solving the standard multiobjective test problems ZDT and DTLZ.

The survey of related works allows to conclude that all standard parallelization strategies of EA have already been implemented on GPU, including the master-slave model ([15], [16], [17], [18], [20]), the cellular model ([21], [22], [23]) and the island model ([24], [25]). The EC technique that has been more used in parallel implementations on GPU is EP. So far there is only one work that was proposed to develop a generic framework or an automatic code generator ([20]) to simplify the implementation of EAs in GPUs.

The reviewed articles show that EA implementations on GPUs systematically obtained high speedup values, thereby drastically reducing the execution time. In particular, cEA implementations obtained speedup values up to $11.5\times$ ([23]), $20\times$ ([21]) and $73\times$ ([22]), respectively, making attractive the study of this model. On the other hand, there have not been proposals of generic framework for EAs implemented on GPUs and therefore our proposal is a novel approach.

IV. PUGACE

PUGACE is a generic framework for implementing cEAs to solve optimization problems on GPUs. The generic approach is in line with MALLBA (a library of skeletons for sequential and parallel optimization algorithms) [27] and JCell (a cellular genetic algorithms framework) [13]. PUGACE is implemented in C, and uses CUDA (version 2.1) to manage the GPU.

PUGACE supports working with different problem encoding, selection policies, crossover and mutation operators, as well as setting different parameters such as population size, number of generations, mutation and crossover probability. The framework includes the implementation of several evolutionary operators, and it can be extended to incorporate additional operators. PUGACE also supports using a local search mechanism to improve the solutions.

The main features of the proposed framework are presented in the following subsections.

A. PUGACE architecture

The design of PUGACE focuses on implementing an extensible and easy to use framework. To achieve the first objective, new evolutionary operators and neighborhood structures can be incorporated to the framework in an easy way. To achieve the second objective, the framework implementation is separated into several modules that encapsulate different functionalities. However, the complete separation was not entirely possible due to some CUDA (version 2.1) limitations, such as the absence of dynamic memory allocation on GPU and the impossibility of defining a device function in a different module than the one that invokes it.

Figure 2 presents a diagram of the PUGACE modules.

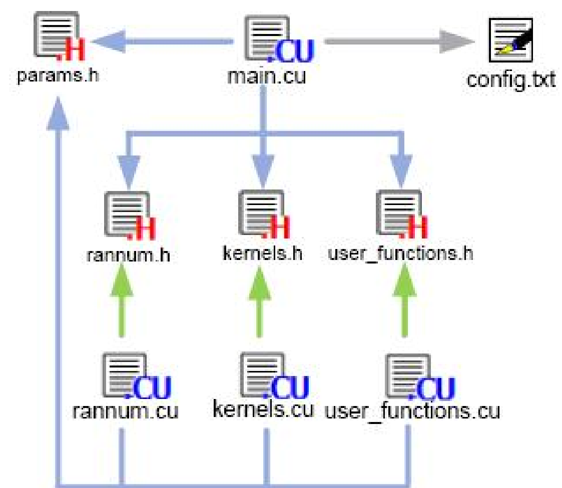

Fig. 2. Diagram of the PUGACE modules.

The main program is *main.cu*, that reads the configuration file, calls the user-defined functions, creates the data structures for the population and the fitness values in the device memory, and calls the GPU procedures. Both the cEA and framework parameters are set in the file *config.txt*.

All the functions and procedures that execute on the GPU are implemented on *kernels.cu*. The methods that are problem-dependant, such as the fitness function and the local search procedure, as well as the methods that are problem-independent, such as new evolutionary operators incorporated to the framework, are implemented in this file. Ideally, these functions and procedures should be implemented in a separate file, but that is not possible due to CUDA limitation already mentioned.

Some additional features, like a procedure for specifically generate the initial population, could be implemented in the file *user_functions.cu*.

Some required parameters of the framework are set in the file *params.h*, due to the CUDA limitations already commented. The procedures that generate the sequence of random numbers are implemented in the file *rannum.cu*.

To instantiate a new problem, at least a fitness function must be implemented in *kernels.cu* and both the cEA and the framework parameters must be configured in *config.txt*.

The PUGACE design follows the next guidelines:

- The population of chromosomes always resides in the device memory, and it is transferred from the CPU to the GPU at the beginning of the algorithm, and viceversa at the end of the algorithm.
- Each individual uses a different execution thread, and threads are organized in blocks of varying size.
- To improve the algorithmic performance, the framework exploits the different GPU memory levels. In particular, the information of the problem is preloaded on constant memory.
- The crossover and mutation operators are applied probabilistically. So, some threads may apply them and others do not, or crossover may occur at different points, which can cause divergent threads within a block. To avoid the thread divergence, the application of crossover and mutation operators is decided at the block level; only one decision per operator is made for each block. The crossover points are also selected at block level, this is to say that the same crossover points are used within a block. On the other hand, the mutation points and the new mutation values are selected independently for each thread.

Designing a highly flexible and general framework usually is in conflict with obtaining implementations that are as efficient as possible. Since this is a first approach to develop the PUGACE framework, the generality of the design is favored over the efficiency. For this reason, some GPUs aspects were not taken into account when designing PUGACE, such as maximizing the utilization of shared block memory and coalescing the access to global memory. Incorporating such aspects in the PUGACE design will imply an improvement in the efficiency of the implementation, so it is planned to study how to take into account this aspects in a future version of the framework.

B. Population and neighborhood structure

The population is arranged in a circular 1-dimensional structure, according to ideas proposed by Baluja [28]. The individuals from both ends of the population are copied to the opposite end to simplify the application of the operators. The number of copied individuals at each end varies depending on the length of the neighborhood structure used. Figure 3 shows an example of a population with four copied individuals (chromosomes 1, 2, N-1, and N). Copying individuals helps avoiding the thread divergence, since all neighbors are close. However, it requires an additional step in each generation in which the copies acquire the values associated to the original individuals.

The population can be randomly initialized or specifically generated. This second option is useful to incorporate a particular heuristic algorithm for the optimization problem. After the population is initialized, the individuals are copied into the device global memory where they reside until the evolutionary process end.

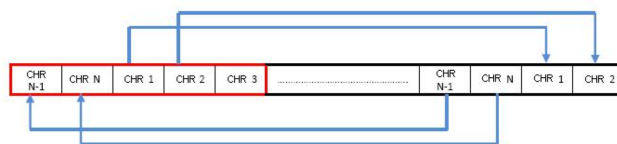

Fig. 3. Circular 1-dimensional population structure.

Neighborhood structure consists of a configurable number of individuals to the left and right. The neighborhood structure length is a parameter of the framework that also determines the number of individuals that must be copied on both ends.

C. Fitness function

The values of fitness function are stored in an auxiliary vector in the same order as the chromosome population. Like the population, the fitness values vector is stored in the device memory.

The fitness function evaluation uses an independent thread for each chromosome of the population, except for copies. Each thread computes the fitness function and stores the value at the corresponding chromosome position of the fitness vector. The fitness function is problem-dependent and must be implemented for each problem.

D. Evolutionary operators

Different alternatives for selection, crossover and mutation operators were included. PUGACE implements a minimal number of different operators; but the framework can be easily extended with new implementations of the operators.

The selection operator is applied for each of the population positions for choosing two individuals for reproduction. Three selection operators are provided by PUGACE. All of them choose the individual at the considered position as one of the mating individuals. According to the operator employed, the other individual could be choose randomly, fitness proportionally, or the best one from the neighborhood.

The crossover operator is applied to recombine the individuals selected for reproduction, obtaining two new solutions. PUGACE provides implementations of the classic one point and two point crossovers, as well as Partially Matched Crossover (PMX) [29] crossover for permutations encoding.

The mutation operator is applied after the reproduction to both offspring. PUGACE provides two different mutation operators. The first one modifies one of the chromosome values with a random value. The second one exchanges two randomly selected values of the chromosome.

PUGACE uses a generational replacement strategy. Each parent is replaced by the best one of its children. The replacement is synchronous, this is to say that all children replace its parents at the same time.

The framework allows incorporating a local search operator to improve the solutions during the evaluation. Obviously,

as such operators are problem-dependent, they have to be implemented for each problem.

E. Random number generation

As the GPU does not provide a pseudo-random numbers generator, two different alternatives are implemented in PUGACE to tackle this problem.

The first alternative is that pseudo-random numbers are generated in the CPU, and they are transferred to the GPU in each generation, taking benefit of the CPU idle times.

The second option is to directly generate pseudo-random numbers in the GPU with a specific algorithm based on a linear congruential method.

V. EXPERIMENTAL RESULTS

This section presents experimental results. It introduces the optimization problem chosen to evaluate the proposed framework, the classical Quadratic Assignment Problem. It also presents details on the cEA parametrization for solving this problem. Finally, the quality and performance results are commented.

A. Quadratic Assignment Problem

Quadratic Assignment Problem (QAP) [30] model many applications in diverse areas such as operations research, parallel and distributed computing, and combinatorial data analysis. QAP is a NP-hard problem.

The problem is to assign N facilities to N locations with costs proportional to the flow between facilities multiplied by the distances between locations. The goal is to assign each facility to a different location such that the total cost is minimized.

This assignment problem can be modeled by two $N \times N$ matrices:

- $A = (a_{ik})$, where a_{ik} is the flow from facility i to facility k;
- $B = (b_{jl})$, where b_{jl} is the distance from location j to location l;

The objective function is defined by Equation 1, where S_n is the set of all permutations of the integers $1, 2, ..., n$. Each product $a_{ik} b_{\varphi(i)\varphi(k)}$ is the transportation cost associated to place the facility i on the location $\varphi(i)$ and the facility k on the location $\varphi(k)$. Each term $\sum_{k=1..n} a_{ik} b\varphi(i)\varphi(k)$ is the total cost for installing i.

$$min_{\varphi \in S_n} \left(\sum_{i=1}^{n} \sum_{k=1}^{n} a_{ik} b_{\varphi(i)\varphi(k)} \right) \quad (1)$$

Instances of the QAP with input matrices A and B are generally denoted by QAP (A, B). The QAPLIB [31] library contains several test sets with instances generated with different criteria: symmetric matrices, asymmetric matrices, randomly generated, rectangular distances, real world data, etc. In the PUGACE evaluation we selected several instances with up to 50 facilities and locations from five different test sets.

B. Problem encoding

A permutation representation for encoding QAP solutions was used. A feasible solution is represented as an integer array; each integer value indicates the location where the facility given by index array position is placed.

The population was initialized by generating random feasible permutations. The evolutionary operators used were proportional selection to the fitness of the neighbors, the PMX crossover operator and the mutation operator that exchanges two randomly selected values.

A local search method was implemented to improve solutions. The local search randomly selects a chromosome position and evaluates all possible exchanges between the selected position and the rest. The exchange that produces the largest increase in the fitness solution is applied. The chosen position is the same for the whole population, but it varies during the generations. For efficiency reason, the local search method is applied every ten generations.

C. Parameters settings

A configuration analysis was carried out to determine the best parameter values for solving the QAP. The parameters considered in the analysis were: population size (512, 1024 and 2048), crossover probability (0.7, 0.8 and 0.9), mutation probability (0.05, 0.1, 0.2) and neighborhood length (2 and 4).

The configuration analysis considered two instances from two different test sets of QAPLIB, $Chr25a$ and $Kra30$. The parameter values were combined following the orthogonality principle. Ten independent executions were run for each different configuration using a stop criterion of reaching 500 generations. From the comparison of the best solution found and the average solution cost for each configuration, it was determined that the best parameter values were: population size = 2048, crossover probability = 0.9, mutation probability = 0.1 and neighborhood length = 4. No specific experiments were conducted to determine the optimum values for the number of threads per block and the number of thread blocks.

The results also showed that there is a direct relationship between the crossover probability value and the runtime of the algorithm. When the mutation probability used is 0.1, the execution time is 5% larger than when using a probability of 0.05, while when using a probability of 0.2 the execution time is 10% larger than when using a mutation probability of 0.1.

The stopping criterion adopted was to reach a specified number of generations. A high limit value for the number of generations was used (5000) trying to reach high quality solutions.

Table I summarizes the parameter values used in this work.

D. Execution platform

The execution platform used was a Pentium dual-core at 2.5 GHz with a 2 GB RAM and an nVidia GeForce 9800 GTX+ graphic card.

TABLE I

CEA PARAMETERS USED FOR SOLVING THE QAP.

population size	2048
generations	5000
thread blocks	32
threads per block	64
crossover probability	0.9
mutation probability	0.1
neighborhood length	4

E. Solutions quality

This subsection presents and discusses results obtained by the cEA implemented on PUGACE for solving fourteen QAP instances.

Table II presents the results obtained by the cEA on ten independent runs for each QAP instance studied. The table includes the number of facilities and locations of the instance, the best known solution, the number of executions that the best know solution was found (# Hits) and the gap respect to the best known solution, defined by Equation 2.

$$Gap = \frac{(Solution - Best\ Known\ Solution)}{(Best\ Known\ Solution)} \quad (2)$$

TABLE II

RESULTS OF CEA FOR SOLVING QAP.

Instance	N	Best known solution	# Hits	Gap (%)
Bur26a	26	5426670	7	0.00
Chr12a	12	9552	8	0.00
Chr15a	15	9896	7	0.00
Chr18a	18	11098	10	0.00
Chr20a	20	2192	8	0.00
Esc16f	16	0	6	0.00
Esc32a	32	130	4	0.00
Had12	12	1652	8	0.00
Had14	14	2724	4	0.00
Had16	16	3720	6	0.00
Lipa30a	30	13178	2	0.00
Lipa30b	30	151426	1	0.00
Lipa40a	40	31538	0	1.05
Lipa50b	50	1210244	1	0.00

The implemented algorithm reached the best known solution in 13 out of 14 of the problem instances considered. Only in one instance the best known solution was not found, but the gap was small (1.05%). The results are acceptable, but they start to degrade when considering larger instances, possibly because the followed approach for solving the QAP was too simple. On the other hand, the population size is relatively smaller than the used by other authors for GPU implementation of EAs. Increasing the population size could contribute to improve the quality of the solutions by increasing the diversity of the population.

The proposed framework showed its potential to easily implement a cEA that obtained good quality solutions for a classical combinatorial optimization problem.

F. Performance results

This subsection presents and compares the computational performance of the cEA implemented on GPU and a sequential cEA implemented on CPU. The tests were conducted in order to evaluate the reductions in runtime that can be obtained by implementing a cEA on PUGACE rather than on a CPU.

Table III presents the execution time of the cEA implemented on GPU, the execution time of the sequential cEA implemented on CPU and the algorithm speedup (sequential execution time / parallel execution time) evaluated in ten independent runs for the QAP instances studied. The speedup values reported consider the total runtime of the algorithm, including the transfer time.

TABLE III

PERFORMANCE RESULTS OF CEA IMPLEMENTED ON GPU AND ON CPU.

Case	GPU implementation time (s)	CPU implementation time (s)	Algorithm speedup
Bur26a	16.961	270.22	15.93
Chr12a	2.559	45.943	17.95
Chr15b	4.802	74.152	15.44
Chr18a	6.092	112.142	18.41
Chr20a	7.853	143.285	18.25
Esc16f	5.905	88.983	15.07
Esc32a	28.06	455.564	16.24
Had12	2.532	46.081	18.20
Had14	3.444	63.82	18.53
Had20	7.963	143.281	17.99
Lipa30a	21.599	385.926	17.87
Lipa30b	21.044	386.632	18.37
Lipa40a	46.852	808.803	17.26
Lipa50b	91.429	1460.298	15.97

The speedup values obtained for the different QAP instances ranged between 15 and 19, showing the potential of the proposed framework for using GPU devices to significantly reduce the runtimes of cEA. Although the speedup values are high, there are several areas for improvement such as maximizing the utilization of shared block memory and the coalescence of the access to the global memory.

Another important result concerning the PUGACE performance was found on the calibration stage. In the GPU implementation, the increase in the population size impacts in a sublinear increase in the execution time. For example, doubling the number of individuals in the population (from 512 to 1024 and from 1024 to 2048) implies runtime increments of less than 10%. Therefore, the effect of increasing the population size in the solutions quality and the algorithm performance should be studied thoroughly.

VI. CONCLUSION AND FUTURE WORK

This work presented PUGACE, a general framework of cEAs implemented on GPU. The aim of our proposal is to significantly reduce the cEAs runtime through exploiting the potential of current multi-core GPUs. This work also includes a study of algorithmic behavior and performance when implementing a cEA for solving the QAP on PUGACE, to show the potential of the proposed framework.

Analyzing the quality and performance results obtained for the QAP, some conclusions can be drawn on the applicability of the proposed framework. It can be noted that PUGACE is a useful tool for implementing cEA for solving optimization problems, and a viable option to easily exploit GPUs. In addition, high values of speedup were obtained, showing that cEA is highly suitable for the implementation on GPUs, obtaining high reductions in the execution time.

Three main areas that deserve further work are the evaluation of the applicability of PUGACE, the extension of the proposed framework, and the experimentation with other hardware configurations.

In the first place, in order to evaluate the applicability and to validate the generality of PUGACE, the framework must be applied to solve other optimization problems. The problems should be carefully chosen assuring that the fitness functions are complex enough to exploit the GPUs potential.

A second issue that deserves further work is the extension of the proposed framework. Currently, the framework only supports linear neighborhood structures. New neighborhoods structures should be implemented, as well as new evolutionary operators should be incorporated to the framework to make PUGACE a more powerful tool. Also, some aspects of the CUDA behavior that were not considered in this work should be taken into account, such as maximizing the utilization of shared block memory and coalescing the access to the global memory.

Finally, further work has to be done to extend the experiment with other hardware configurations. The framework was designed to encapsulate the GPU particularities, but the experiments performed in this work used only one GPU model. New experiments should be made on different devices with diverse features to validate the implementation of the framework. Complementarily, it is planned to investigate on how to extend PUGACE to support different platforms that involve GPU usage, such as one CPU with multiple graphic cards (i.e. Tesla) and a cluster of PCs with graphic cards.

ACKNOWLEDGMENT

The authors would like to thank Ph.D. Sergio Nesmachnow for his help. Also, we thank the anonymous reviewers for their valuable suggestions that improved this work.

REFERENCES

[1] "Cg website," http://developer.nvidia.com/page/cg_main.html. Accessed on May 2010.

[2] "OpenGL website," http://www.opengl.org/. Accessed on May 2010.

[3] "DIRECTX website," http://developer.nvidia.com/page/directx.html. Accessed on May 2010.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Transactions on Graphics*, vol. 23, pp. 777–786, 2004.

[5] "Sh website," http://libsh.org/. Accessed on May 2010.

[6] "PyGPU website," http://www.cs.lth.se/home/Calle_Lejdfors/pygpu/. Accessed on May 2010.

[7] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," in *ASPLOS-XII: Proceedings of the 12th Int. Conf. on Architectural support for programming languages and operating systems*. ACM, 2006, pp. 325–335.

[8] "ATI Stream Technology website," www.amd.com/stream. Accessed on May 2010.

[9] "CUDA website," http://www.nvidia.com/object/cuda_home_new.html. Accessed on May 2010.

[10] "OpenCl website," http://www.nvidia.com/object/cuda_opencl_new.html. Accessed on May 2010.

[11] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[12] "GPGPU website," http://gpgpu.org. Accessed on May 2010.

[13] E. Alba and B. Dorronsoro, *Cellular Genetic Algorithms*. Operations Research/Computer Science series, Springer, 2008.

[14] "GPGPGPU website," http://www.gpgpgpu.com. Accessed on May 2010.

[15] M. Wong, T. Wong, and K. Fok, "Parallel evolutionary algorithms on graphics processing unit," in *The 2005 IEEE Congress on Evolutionary Computation*, vol. 3, 2005, pp. 2286–2293.

[16] M. Wong and T. Wong, "Parallel hybrid genetic algorithms on Consumer-Level graphics hardware," in *The 2006 IEEE Congress on Evolutionary Computation*, 2006, pp. 2973–2980.

[17] ——, "Implementation of parallel genetic algorithms on graphics processing units," in *Intelligent and Evolutionary Systems*, 2009, pp. 197–216.

[18] S. Harding and W. Banzhaf, "Fast genetic programming and artificial developmental systems on GPUs," in *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)*. IEEE Computer Society, 2007, p. 2.

[19] ——, "Genetic programming on GPUs for image processing," *International Journal of High Performance Systems Architecture*, vol. 1, no. 4, pp. 231 – 240, 2008.

[20] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet, "Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA," in *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2009, pp. 1403–1410.

[21] Q. Yu, C. Chen, and Z. Pan, "Parallel genetic algorithms on programmable graphics hardware," in *Advances in Natural Computation*, 2005, pp. 1051–1059.

[22] J. Li, X. Wang, R. He, and Z. Chi, "An efficient fine-grained parallel genetic algorithm based on GPU-Accelerated," in *IFIP International Conference on Network and Parallel Computing Workshops*, 2007, pp. 855–862.

[23] J. Li, L. Zhang, and L. Liu, "A parallel immune algorithm based on fine-grained model with gpu-acceleration," in *Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control*. IEEE Computer Society, 2009, pp. 683–686.

[24] S. Tsutsui and N. Fujimoto, "Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study," in *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2009, pp. 2523–2530.

[25] T. Lewis and G. Magoulas, "Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs," in *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2009, pp. 1379–1386.

[26] M. Wong, "Parallel multi-objective evolutionary algorithms on graphics processing units," in *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2009, pp. 2515–2522.

[27] E. Alba, G. Luque, J. García, G. Ordoñez, and G. Leguizamón, "Mallba; a software library to design efficient optimisation algorithms," *International Journal of Innovative Computing and Applications*, vol. 1, no. 1, pp. 74–85, 2007.

[28] S. Baluja, "A massively distributed parallel genetic algorithm (mdpga)," Carnegie Mellon University, Tech. Rep. CMU-CS-92-196R, 1992.

[29] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[30] E. Loiola, N. de Abreu, P. Boaventura-Netto, P. Hahn, and T. Querido, "A survey for the quadratic assignment problem," *European Journal of Operational Research*, vol. 176, no. 2, pp. 657–690, 2007.

[31] "Qaplib - a quadratic assignment problem library." http://www.seas.upenn.edu/qaplib/. Accessed on May 2010.